2. INCREMENTAL DEVELOPMENT (SPIRAL PROCESS MODEL)

"All things are a combination of earth, fire, water, and software."
-Adapted from Empedocles, 495-435BC

As explained in the *CVISN Guide to Program and Project Planning* [4], software is different and therefore its development must be managed differently. In the engineering profession this has led to a new incremental product development process model known as the "spiral".

2.1 What is a Development Model?

A product development process model is a mental picture – a conceptual framework that serves as an organizing principle for interrelated activities. The notion of the spiral model as a fundamental principle underlying CVISN phase planning and tracking is so vitally important that we devote a chapter to it.

2.2 Linear Development Model Doesn't Work for Software

The linear model (or "waterfall" model because of its shape when drawn) of product development (see Figure 2–1) worked well for centuries, but fails for today's software-intensive, behaviorally complex systems [10, 11]. Some problems with the linear model are:

- Users don't know precisely what they need from an automated system until they begin to see it in operation, so up-front requirements cannot be adequate.
- There is no opportunity for design re-direction based upon user experience.
- Technology evolves rapidly, thereby making earlier choices less effective or even obsolete.
- The time frame (typically 3 years) is too long from concept to operation, and consequently stakeholder commitment evaporates.
- There is a high risk of never having anything operational.

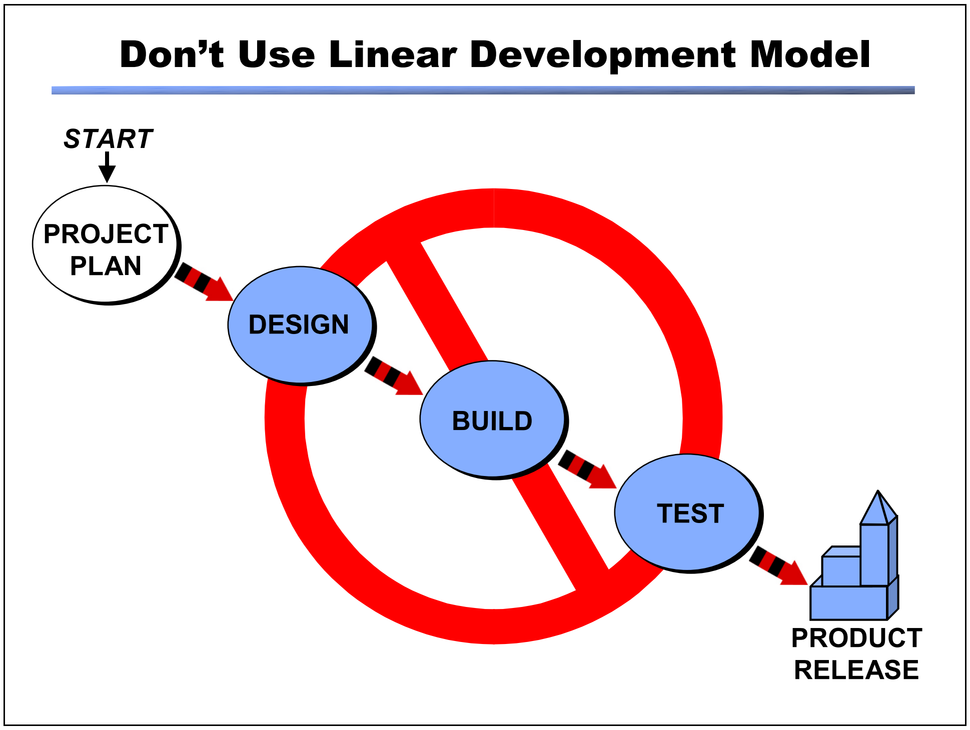

Figure 2-1. Linear Development Model is Inadequate for Software-Intensive Products

2.3 Spiral Development Model Works Well

The essence of the spiral model is first to establish a baseline plan and an overall vision of the architecture; and then to deploy the product incrementally by successive iterations through design, build, test, and next-phase planning.

The spiral development model deals with the shortcomings of the linear development model:

- Users react when they see the system in operation.
- Each turn of the spiral (or phase) is an opportunity for design re-direction based upon user experience.
- Time frame of each turn of the spiral is typically 3-6 months, and therefore stakeholder commitment is nourished by constant progress.
- Each phase is an opportunity to absorb new technology.
- After each turn of the spiral at least that much is operational.

Here's an analogy: suppose your organization began awarding annual bonuses of $15,000 and your family decided to dedicate it each year for ten years to turning your existing house into your dream house. Where do you begin?

The analogy commences in Phase 1 (Figure 2–2) by means of consultations with the architect who prepares a complete site plan. Do you then simultaneously start digging a pool, paneling the family room, and penetrating the exterior wall for a fireplace? NO! What would happen if next year you had to move, or if the bonuses dry up? You'd end up with a big hole in the yard, a half-paneled family room, and a piece of plywood on the exterior wall. Instead you would fully complete one portion of the vision as your first deliverable product for Phase 1 – say the fireplace.

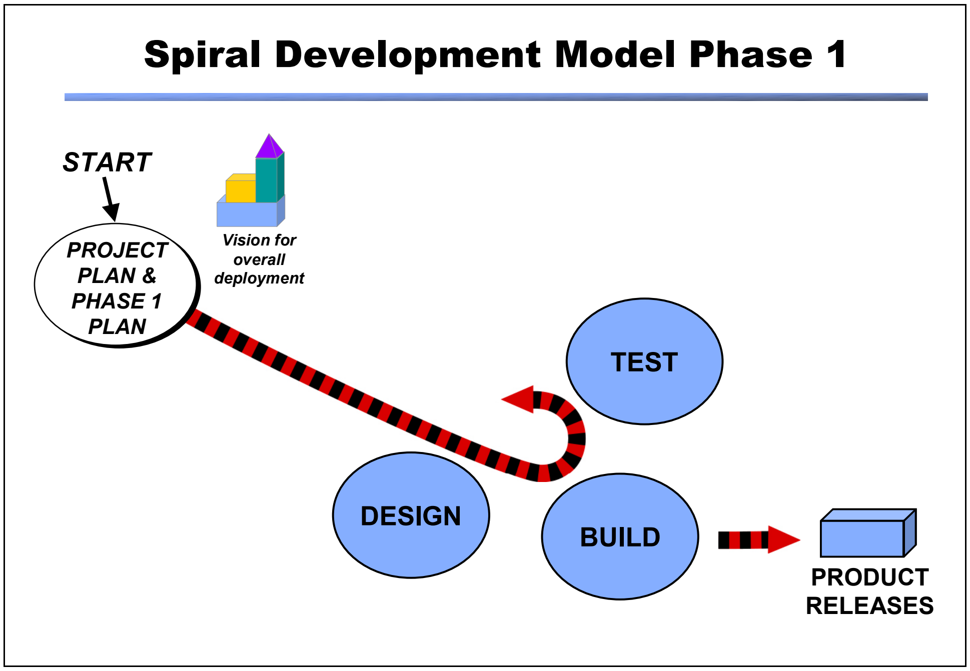

Figure 2-2. First Iteration Through the Spiral Model

For an illustrative CVISN project: at the end of Phase 1 your state architecture has been firmly established, and the essential elements of the computing and networking infrastructure have been procured. You might have hardware (such as servers) but no redundancy yet (such as hot-spare servers); operating systems (such as Windows or Unix); a database management system (such as Oracle or Sybase); network protocols (such as TCP/IP or SNA); and interface standards (such as EDI or XML).

Continuing the dream house analogy: in the second year (Figure 2–3) you are fortunate to receive another bonus. The family's priorities have changed, and now they want the basketball court sooner than the swimming pool. The architect updates the site plan per new safety regulations that now require a fence, and a contractor paves the basketball court. Good thing for you the hole for the pool wasn't already dug, because then there wouldn't have been space for the required fencing.

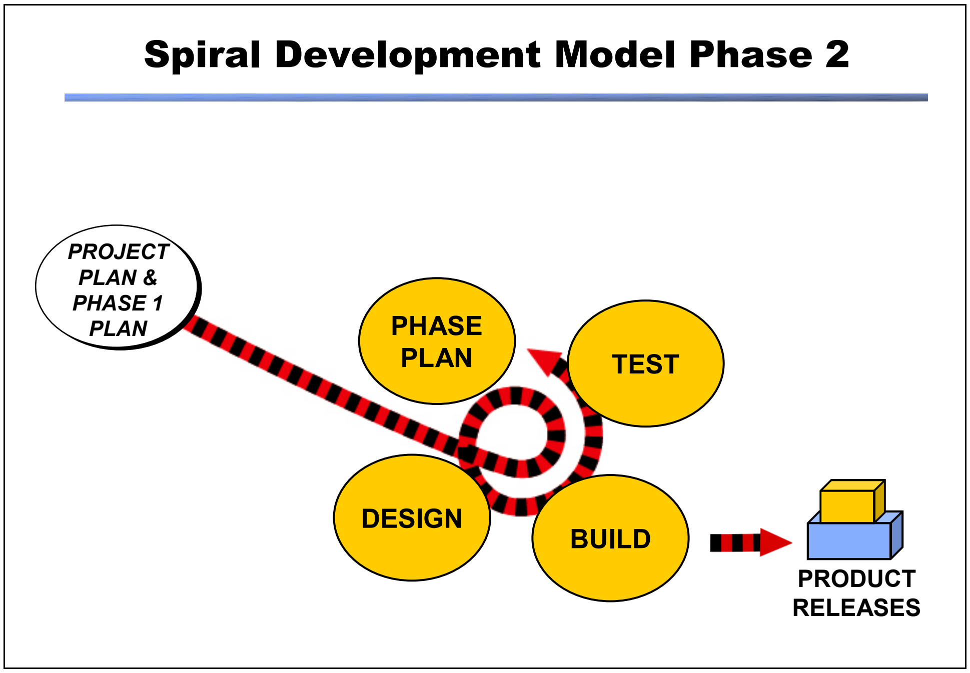

Figure 2-3. Second Iteration Through the Spiral Model

In the illustrative CVISN project: at this point perhaps a prototype user's Carrier Automated Transaction (CAT) software package, along with a prototype state Credentialing Interface (CI) server, would be functional in a laboratory setting – running a few basic user-oriented threads such as the capability to submit an International Registration Plan (IRP) supplemental application but without connection to the IRP processing center.

Continuing with the dream house analogy (Figure 2–4): bad news, bad year – bonuses are much smaller for everyone. You alter the priorities within the constraints, and decide the attic steps go in this year, hopefully the pool next year. Happily the fireplace and basketball court are fully functional.

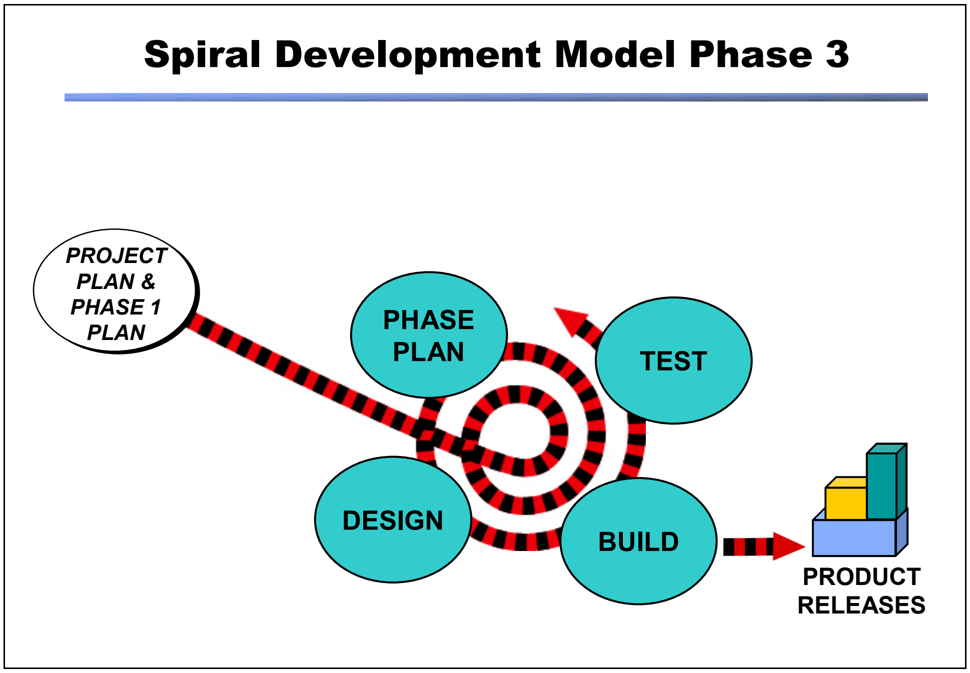

Figure 2-4. Third Iteration Through the Spiral Model

In the illustrative CVISN project: at this point, for example, the CAT hardware would be deployed to a user's site; and the CI hardware deployed in the state's server room with full physical security, firewall, and an uninterruptible power supply. However, there is not yet redundant hardware. Transaction functionality would be demonstrable without embarrassment (such as an end-to-end IRP supplemental application, but perhaps without actually printing the final cab card).

Wrapping up the dream house analogy (Figure 2–5): good news this year, bonuses are higher than ever –but this will be their last year. You complete the pool and fencing, abandon the bowling alley.

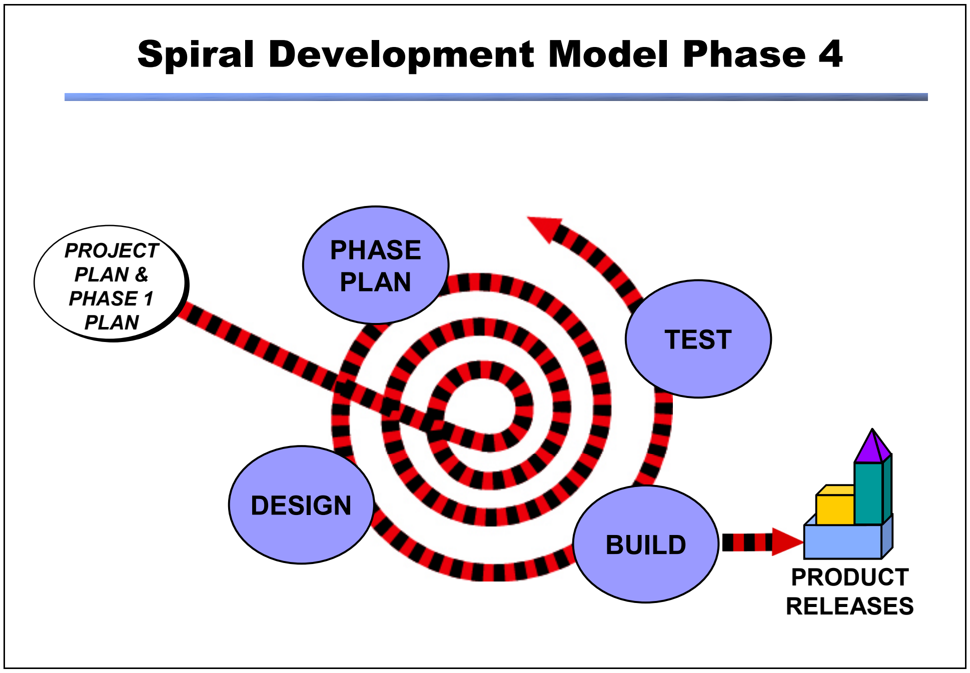

Figure 2-5. Final Iteration Through the Spiral Model

In the illustrative CVISN project: all hardware and infrastructure would be in place, such as a multi-year contract for AAMVAnet and complete redundancy for mission-critical hardware. Complete end-to-end functionality could be demonstrated now, including printing a credential and receiving electronic payment. Perhaps the number of credentials types is still limited, yet the transactions would be end-to-end and useful.

The key point is that after every turn of the spiral there is demonstrable functionality.

In the above approach to development the system is deployed incrementally by successive iterations through design, build (creation), test, and next-phase planning. Detailed requirements analysis and design occur in each phase. This carries the risk that some “hard” requirement is identified at the detailed level that cannot be accommodated by the baseline top-level design. The program-level Configuration Control Board should address such problems. The advantage of the spiral approach is that some capability is available at the end of each phase, and end-users see what the developers are going to provide. That early end-user evaluation allows their feedback to influence future phases. Also, in each phase the designers have an opportunity to take advantage of new technologies.

2.4 What is a Software "Build"?

In the software development world "build" is not only a verb but also a noun used to describe a software entity. The *IEEE Standard Glossary of Software Engineering Terminology* [23] defines a "build" as **an operational version of a system or component that incorporates a specified subset of the capabilities that the final product will provide**.

The notion of builds is useful for phased development and deployment, but usage requires rigorous record-keeping in order to know exactly what is "out there". In the sections below we consider a "build" as an integration of particular versions of products into a working system.

2.5 Configuration Management During Phases and Builds

Figure 2–6 shows the relationships among program phases, project builds, and product versions. Each program phase is associated with a set of defined operational builds within each project. Each project's build is a set of specific versions of products ("releases") that have been integrated and tested together, to provide specified capabilities for that project. Every product release has a version number and an associated description. All of the preceding are under configuration management.

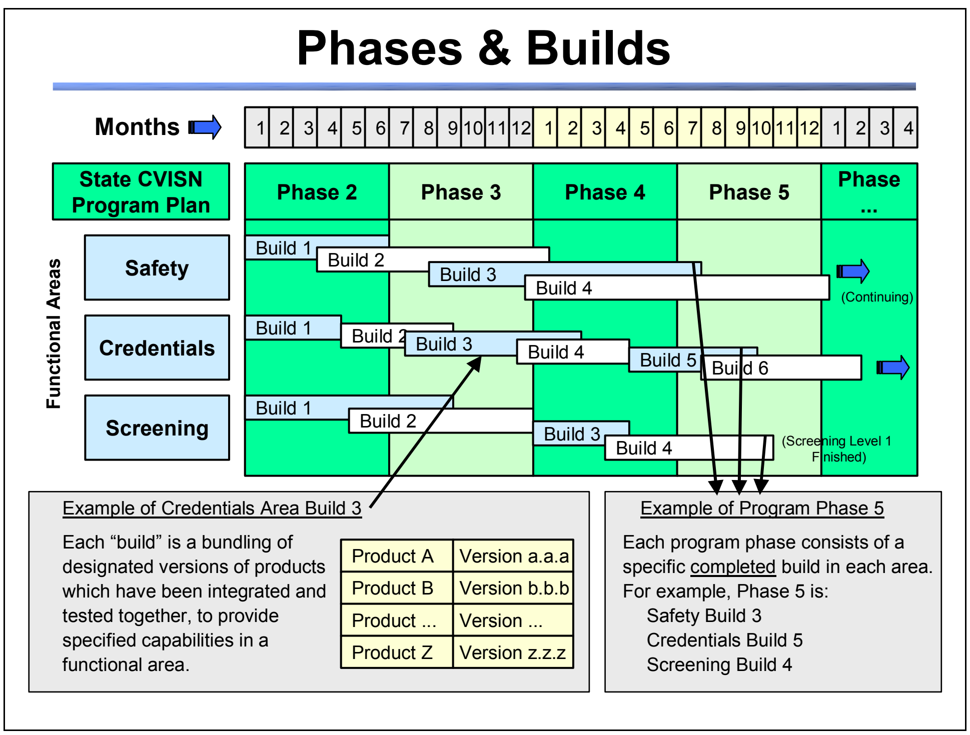

Figure 2-6. Relationships Among Phases, Builds, and Product Versions

A rigorous configuration management process notifies all concerned parties how to handle change requests and approvals, and how to control hardware modifications and software versions. Configuration management is a discipline applying technical and administrative direction and surveillance to identify and document characteristics of every item designated to be under configuration control.

As you make incremental deliveries of products, you need to be able to identify all the components in each delivery, and keep track of problems detected in one phase so that you can control how and when proposed resolutions are implemented.

Figure 2–7 shows how individual products might be identified. Releasing a product version usually means making it available to users for testing or operations. Keeping track of versions of different products makes it possible to identify what components are used to achieve each build's capabilities within a project (Safety Project is shown in this example.) Be sure to establish some naming convention for versions that everyone understands and can apply consistently.

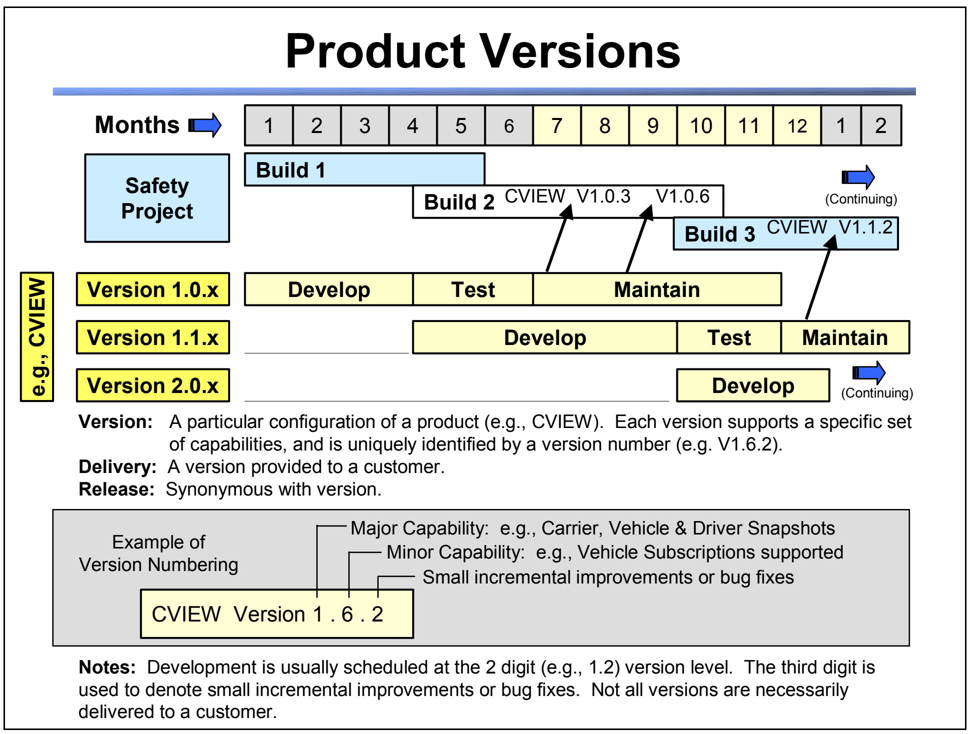

Figure 2-7. Version Identification is Part of Configuration Management

The *IEEE Standard Glossary of Software Engineering Terminology* [23] defines **"version"** as **a release of a computer software configuration item** – meaning that the version number uniquely identifies a configuration-controlled software entity and distinguishes it from similar versions released earlier or later than the one you hold in your hand. This matters when you need to know <u>exactly</u> what the software is supposed to do or when you are calling the manufacturer for help. Figure 2–7 is consistent with that published standard; however you are likely to encounter different usages by various vendors and within other organizations.

By their nature **these terms are used recursively**: one person's system (top-level deliverable) is another person's component (minor element of a larger deliverable system). There can be builds of versions of builds of versions. Although it may seem potentially confusing, it is easily resolved if you explicitly point out the level in the product hierarchy you are referring to.

This Page Intentionally Blank